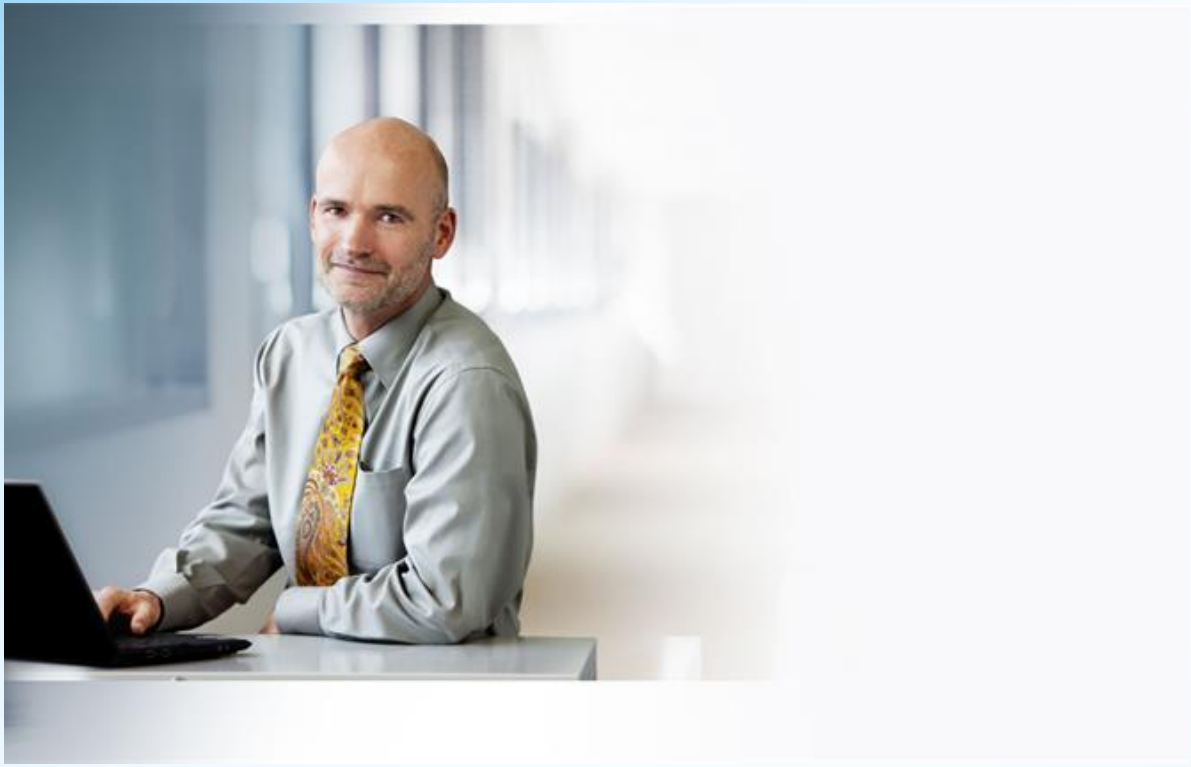
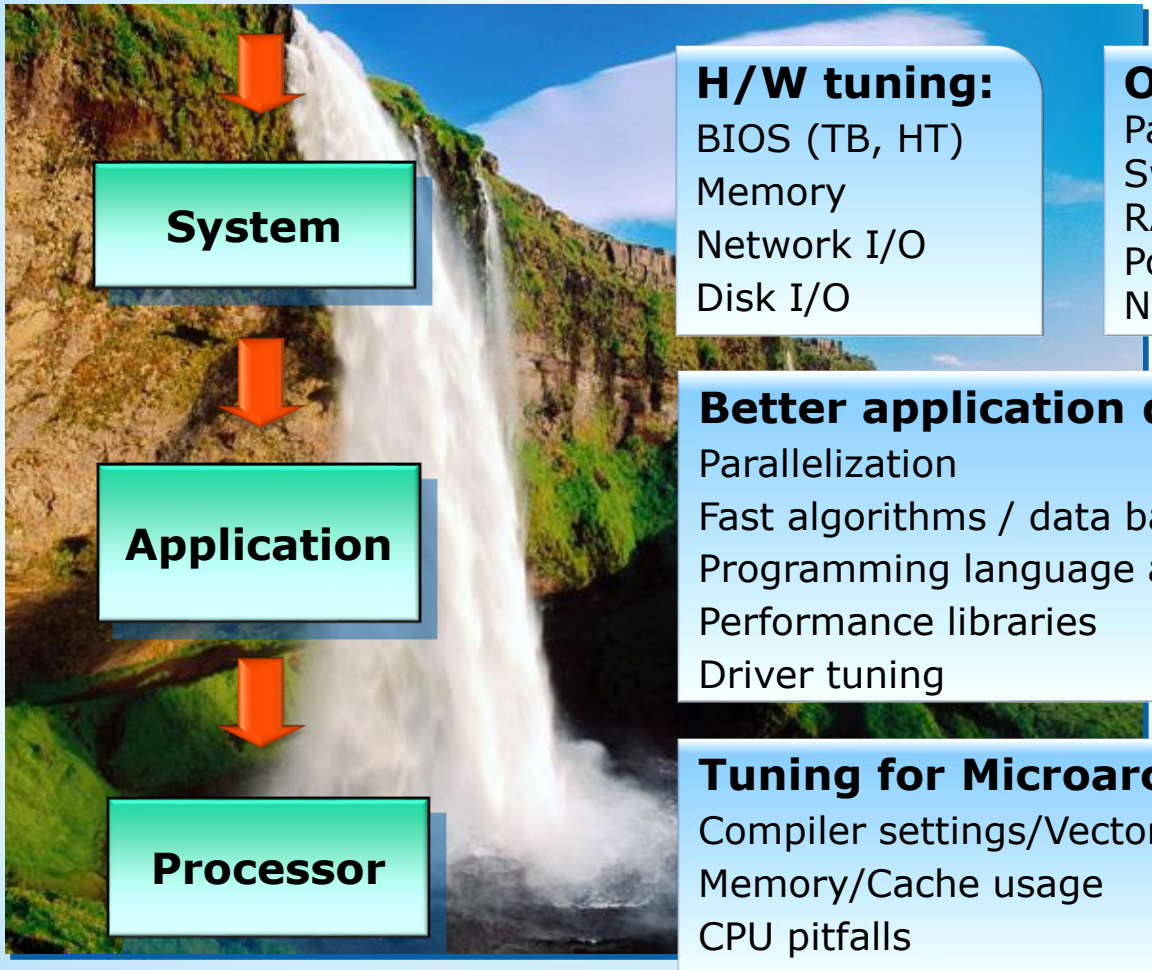


A Performance Tuning Methodology: From the System Down to the Hardware – Diving Deeper

Jackson Marusarz
Intel Corporation
ATPESC 2014

Optimization: A Top-down Approach



H/W tuning:
BIOS (TB, HT)
Memory
Network I/O
Disk I/O

OS tuning:
Page size
Swap file
RAM Disk
Power settings
Network protocols

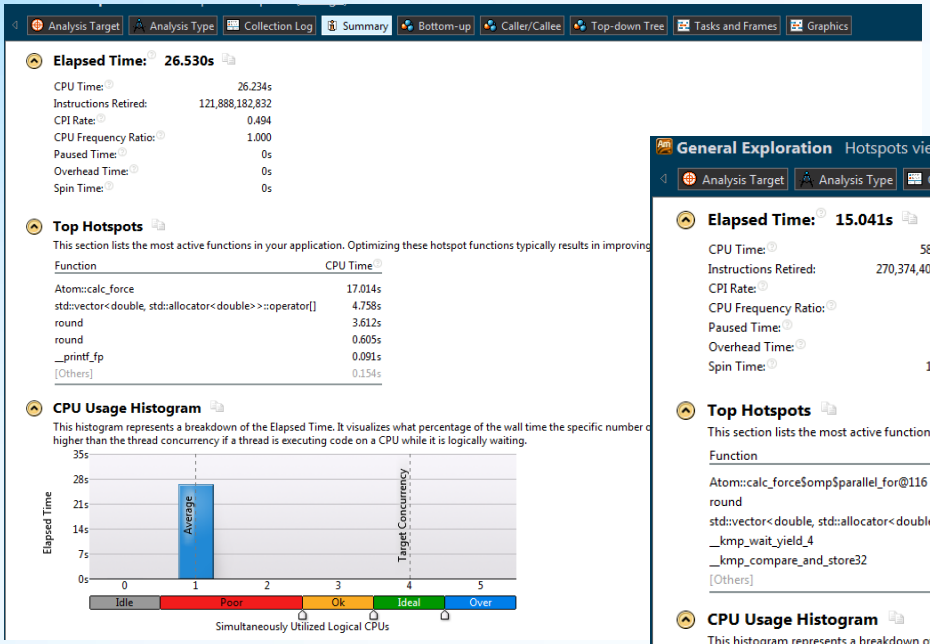
Better application design:
Parallelization
Fast algorithms / data bases
Programming language and RT libs
Performance libraries
Driver tuning

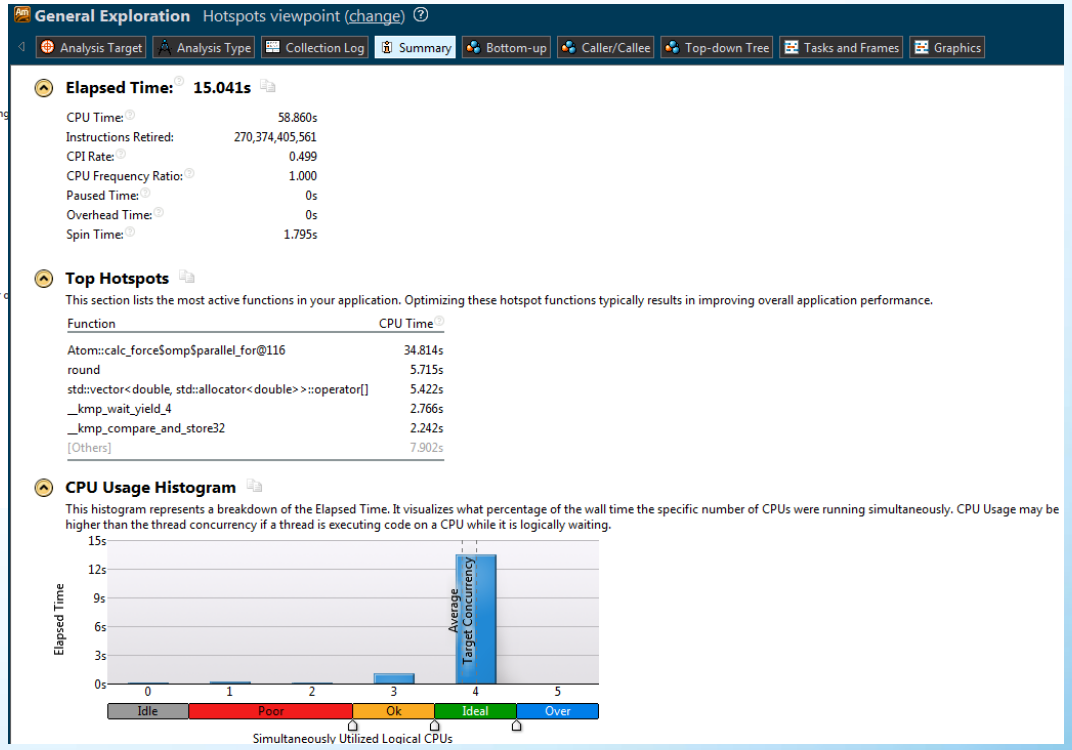
Tuning for Microarchitecture:
Compiler settings/Vectorization
Memory/Cache usage
CPU pitfalls

OS, System **Expertise** SW/uArch

Performance Tuning – Diving Deeper

Perform System and Algorithm tuning first





This presentation uses screenshots from Intel® VTune™ Amplifier XE
The concepts are widely applicable

Algorithm Tuning
A Few Words

- There is no one-size fits all solution to algorithm tuning
- Algorithm changes are often incorporated into the fixes for common issues
- Some considerations:
  - **Parallelizable** and **scalable** over fastest serial implementations
  - Compute a little more to save memory and communication
  - Data locality -> vectorization

Compiler Performance Considerations

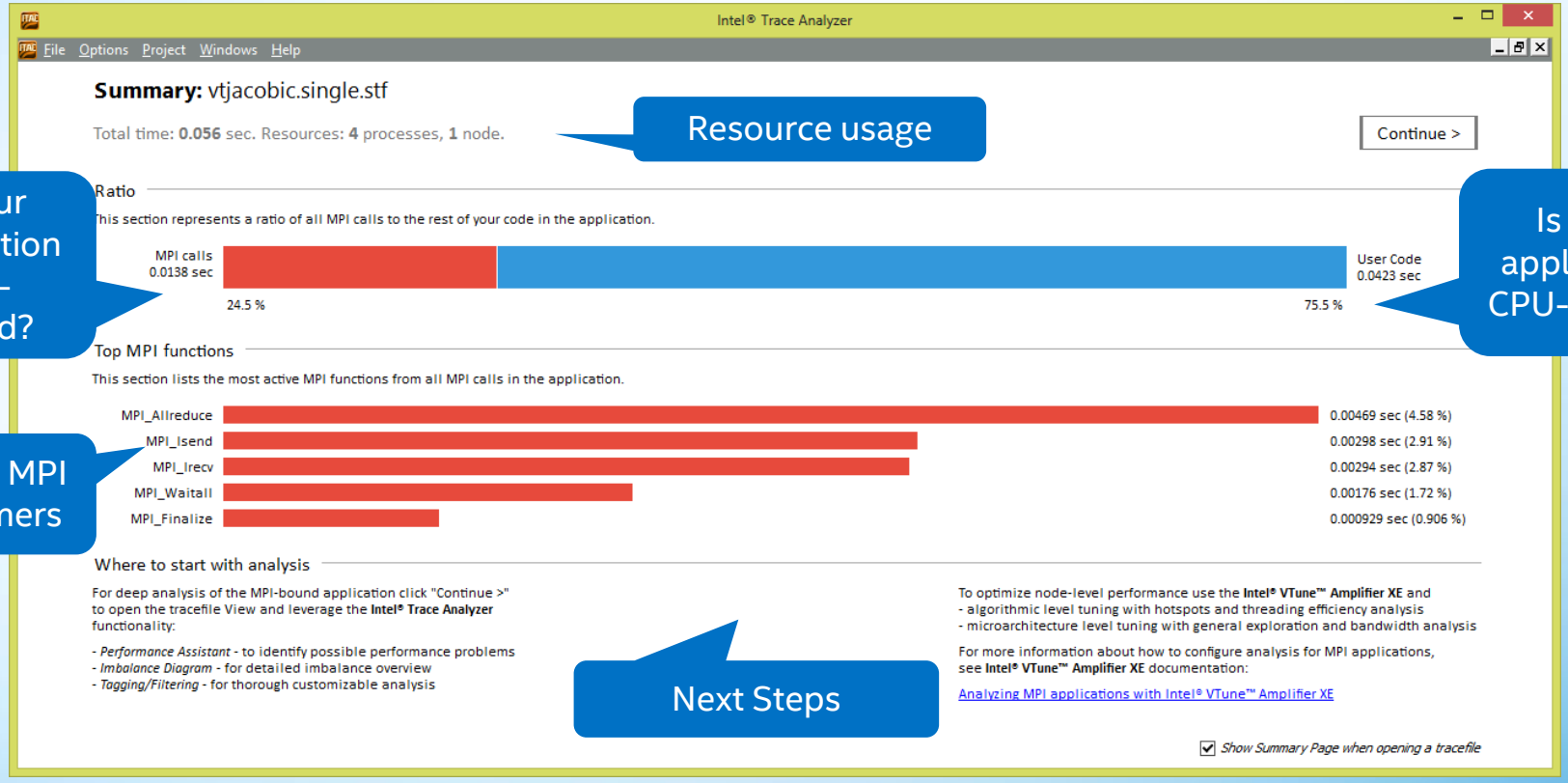
Feature	Flag
Optimization levels	-O0, O1, O2, O3
Vectorization	-xHost, -xavx, etc…
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program **warning: -fast def'n changes over time	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)
Automatic parallelization	-parallel

- Compilers can provide considerable performance gains when used intelligently
- Consider compiling hot libraries and routines with more optimizations
- Always check documentation for accuracy effects
- This could be a day-long talk on its own

This is from the Intel compiler reference, but others are similar

MPI Tuning

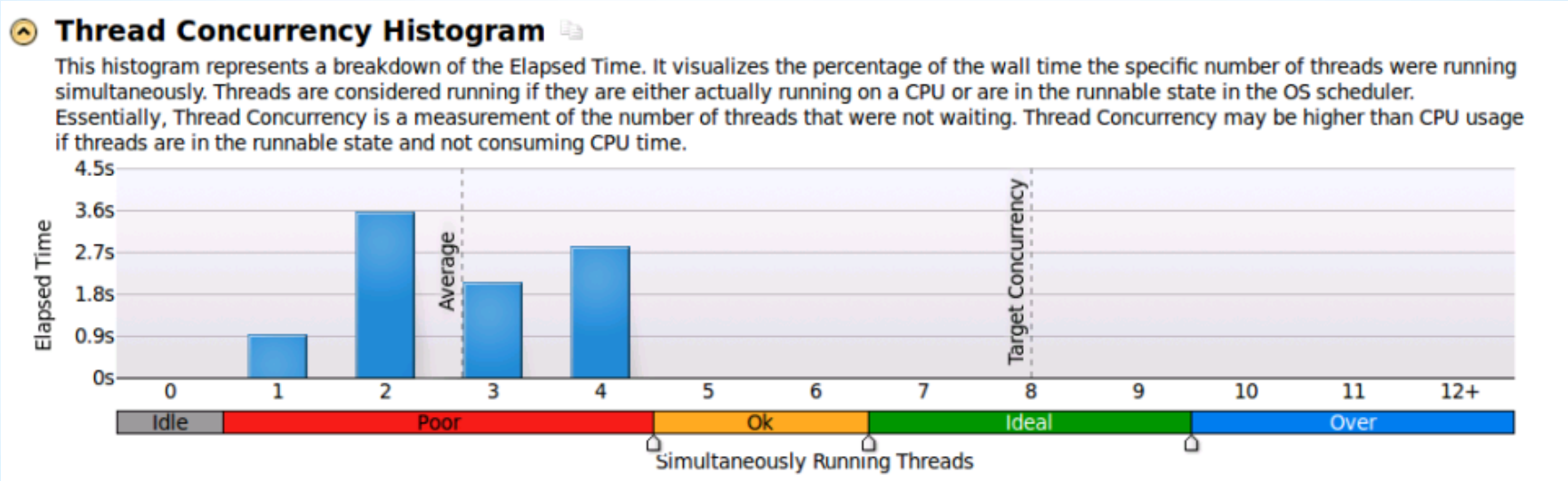
- Find the MPI/OpenMP sweet spot
- Determine how much memory do your ranks/threads share
- Communication and synchronization overhead



Intel® Trace Analyzer and Collector: http://intel.ly/traceanalyzer-collector

Common Scaling Barriers

- Static Thread Scheduling
- Load Imbalance
- Lock Contention



You paid for the nodes, so use them!

Static Thread Scheduling

- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

```
…
NUM_THREADS = 4;
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Static Thread Scheduling

- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

```
…
NUM_THREADS = 4;
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Static Thread Scheduling

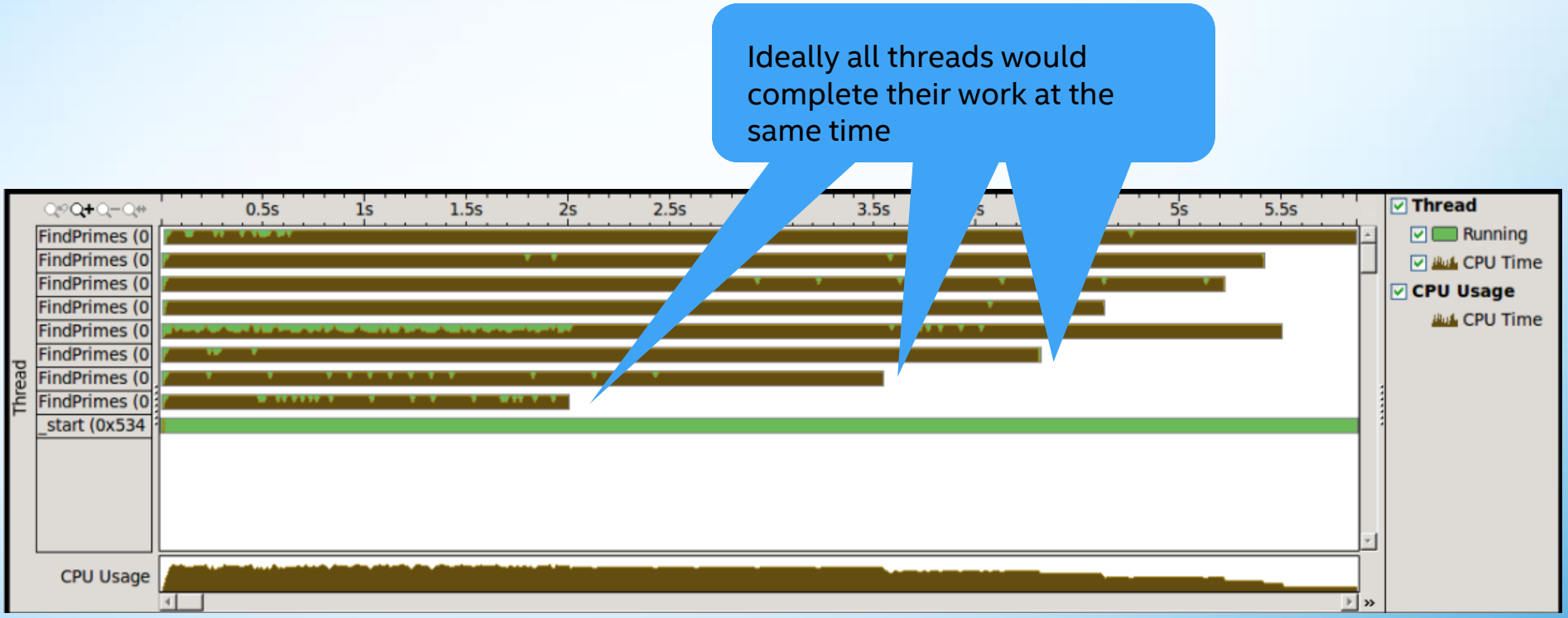
- Statically determining thread counts does not scale
  - Core counts are trending higher
  - Designs must consider future hardware
  - Commonly found in legacy applications

Create Threads Dynamically - `NUM_THREADS = get_num_procs();`

```
pthread_t threads[NUM_THREADS];
int rc;
long t;
int chunk = limit/NUM_THREADS;
for(t=0;t<NUM_THREADS;t++){
  range *r = new range();
  r->begin = t*chunk;
  r->end = t*chunk+chunk-1;
  rc = pthread_create(&threads[t], NULL, FindPrimes, (void *)r);
}
…
```

Load Imbalance

- Dynamically determining thread count helps… but isn't a silver bullet
  - Workload distribution must be intelligent
  - Threads should be kept busy
  - Maximize hardware utilization



Load Imbalance

- Dynamically determining thread count helps… but isn't a silver bullet
  - Workload distribution must be intelligent
  - Threads should be kept busy
  - Maximize hardware utilization

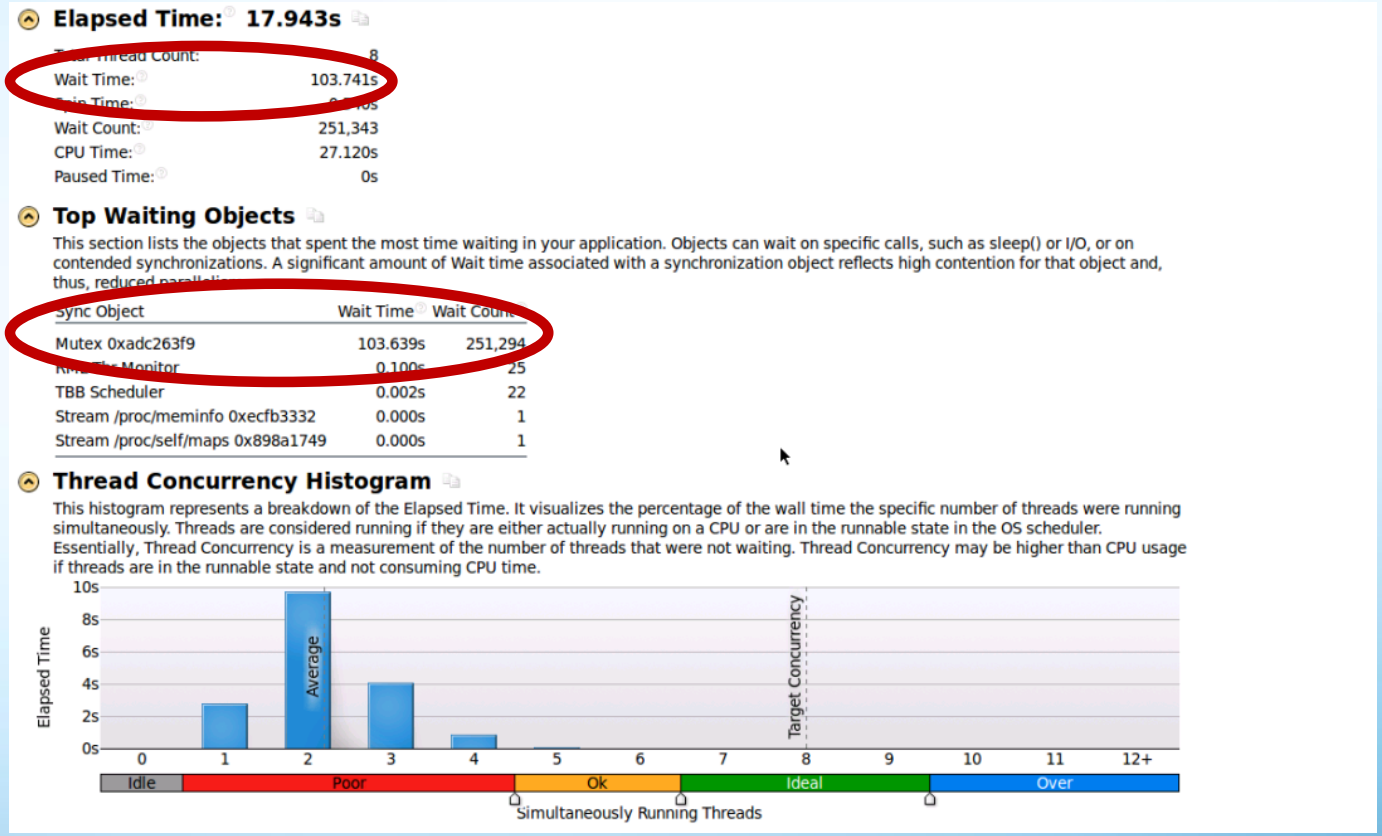
The key to balancing loads is to use a threading model that supports tasking and work stealing

Some examples:

- OpenMP* dynamic scheduling
- Intel Threading® Building Blocks
- Intel® Cilk™ Plus

Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains
  - Numerous choices for where and how to synchronize



Elapsed Time: 17.943s

Wait Time:	103.741s
Wait Count:	251,343
CPU Time:	27.120s
Paused Time:	0s

Top Waiting Objects

This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on contended synchronizations. A significant amount of Wait time associated with a synchronization object reflects high contention for that object and, thus, reduced parallelism.

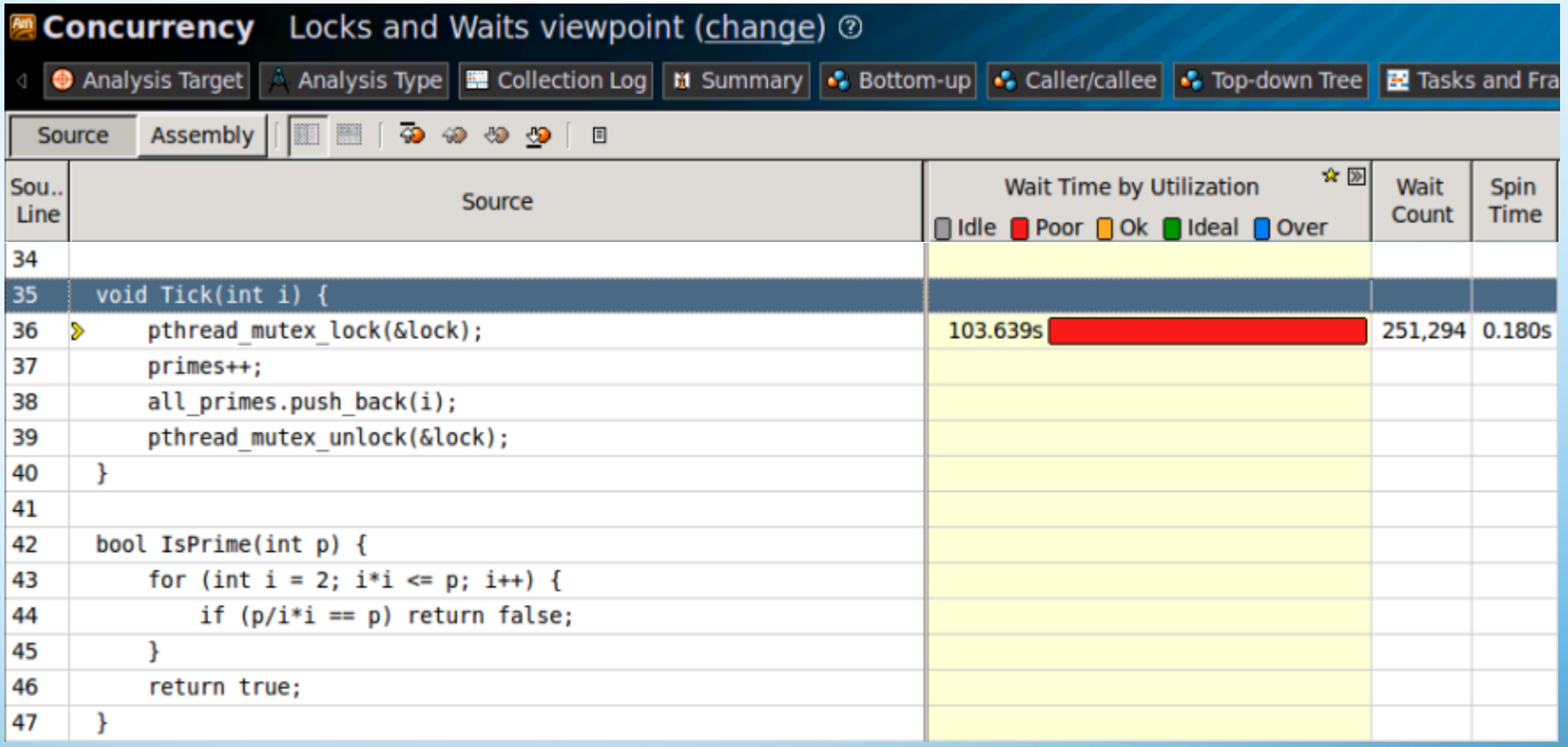
Sync Object	Wait Time	Wait Count
Mutex 0xadc263f9	103.639s	251,294
TBB Scheduler	0.002s	22
Stream /proc/meminfo 0xecfb3332	0.000s	1
Stream /proc/self/maps 0x898a1749	0.000s	1

Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains
  - Numerous choices for where and how to synchronize



Lock Contention

- A well balanced application can still suffer from shared-resource competition
  - Synchronization is a necessary component
  - Excessive overhead can destroy performance gains
  - Numerous choices for where and how to synchronize

Some solutions to consider:

- Lock granularity
  - Access overhead vs. wait time

- Using lock free or thread safe data structures

```
tbb::atomic<int> primes;
tbb::concurrent_vector<int> all_primes;
```

- Local storage and reductions

Microarchitectural Tuning

- Intel uArch specific tuning
- After high-level changes look at PMUs for more tuning
  - Find tuning guide for your hardware at www.intel.com/vtune-tuning-guides
- Every architecture has different events and metrics
- We try to keep things as consistent as possible
- Start with the **Top-Down Methodology**
  - Integrated with the tuning guides

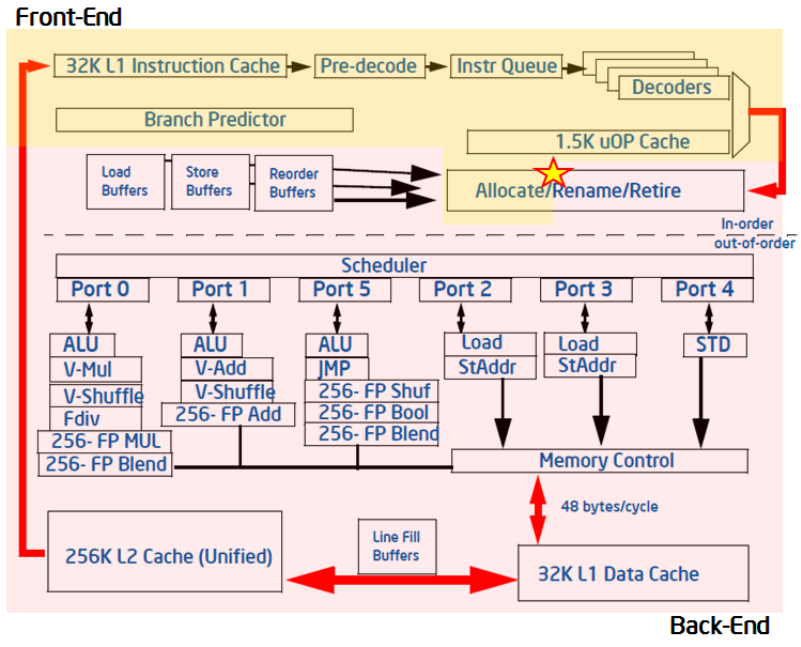
Introduction to Performance Monitoring Unit (PMU)

- Registers on Intel CPUs to count architectural events
  - E.g. Instructions, Cache Misses, Branch Mispredict
- Events can be counted or sampled
  - Sampled events include Instruction Pointer
- Raw event counts are difficult to interpret
  - Use a tool like VTune or Perf with predefined metrics

Background

Hardware Definitions

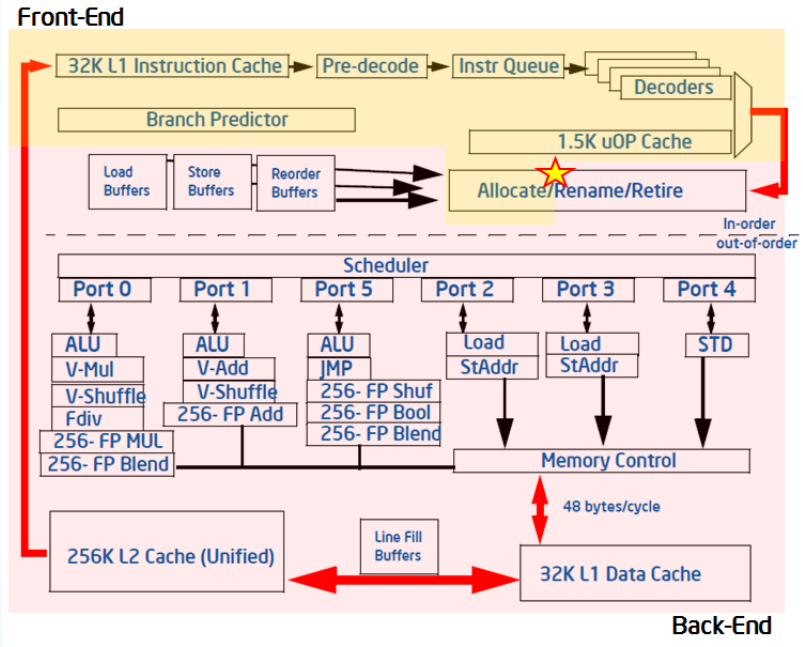
- Front-end:
  - Fetches the program code
  - Decodes them into low-level hardware operations - micro-ops (uops)
  - uops are fed to the Back-end in a process called allocation
  - Can allocate 4 uops per cycle
- Back-end:
  - Monitors when a uop's data operands are available
  - Executes the uop in an available execution unit
  - The completion of a uop's execution is called retirement, and is where results of the uop are committed to the architectural state
  - Can retire 4 uops per cycle
- Pipeline Slot:
  - Represents the hardware resources needed to process one uop



Background

Hardware Definitions

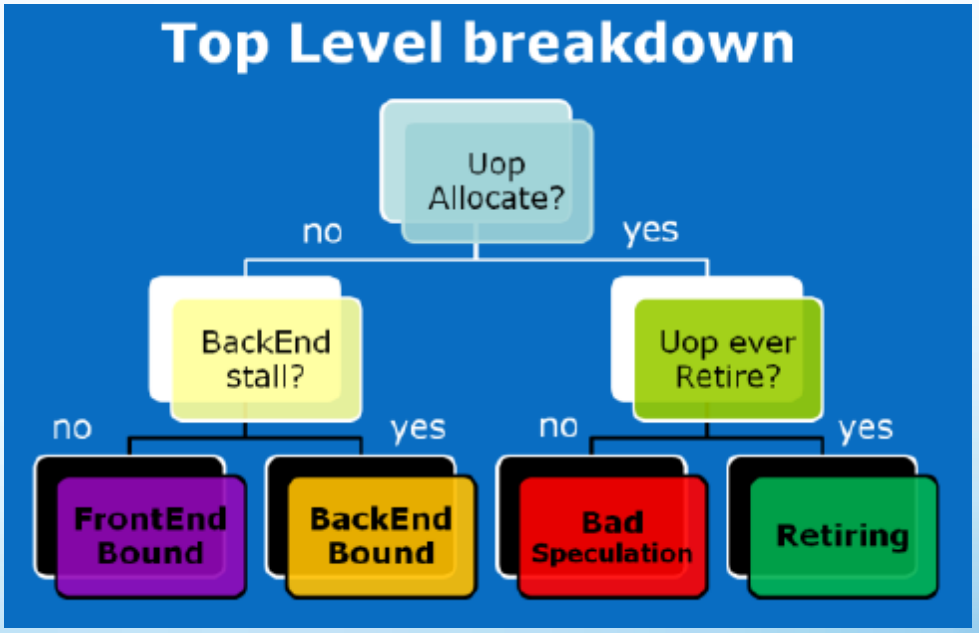
- Front-end:
  - Fetches the program code
  - Decodes them into low-level hardware operations - micro-ops (uops)
  - uops are fed to the Back-end in a process called allocation
  - Can allocate 4 uops per cycle
- Back-end:
  - Monitors when a uop's data operands are available
  - Executes the uop in an available execution unit
  - The completion of a uop's execution is called retirement, and is where results of the uop are committed to the architectural state
  - Can retire 4 uops per cycle
- Pipeline Slot:
  - Represents the hardware resources needed to process one uop



Therefore, modern "Big Core" CPUs have 4 "Pipeline Slots" per cycle

The Top-Down Characterization

- Each pipeline slot on each cycle is classified into 1 of 4 categories.
- For each slot on each cycle:



The Top-Down Characterization

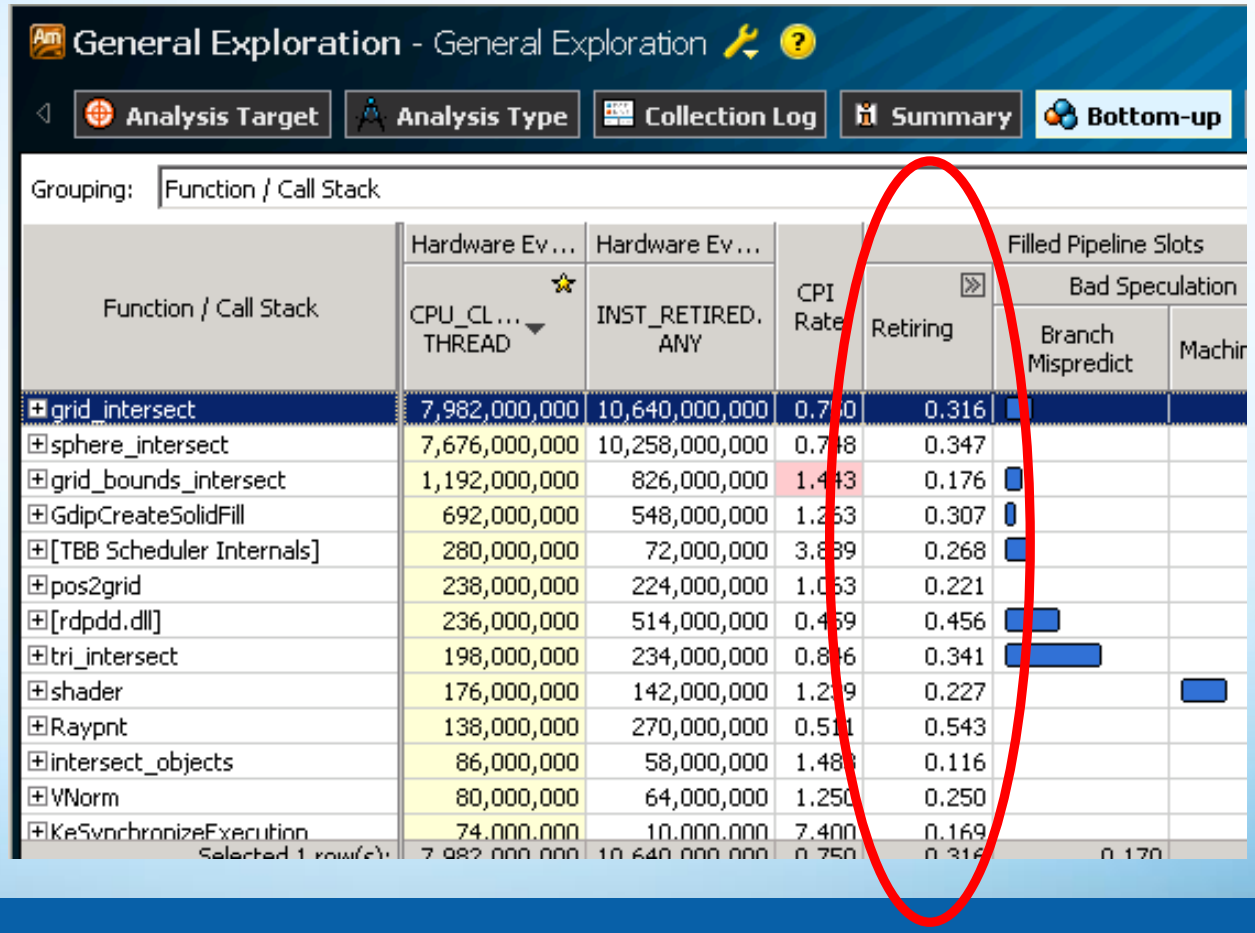
- Determines the hardware bottleneck in an application
- Sum to 1.0
- Unit is “Percentage of total Pipeline Slots”
- This is the core of the new Top-Down characterization
- Each category is further broken down depending on available events
- Top-Down Characterization White Paper
  - http://software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues

Tuning Guide Recommendations

	Expected Range of Pipeline Slots in this Category, for a Hotspot in a *Well-tuned:*		
Category	**Client/ Desktop application**	**Server/ Database/ Distributed application**	**High Performance Computing (HPC) application**
Retiring	20-50%	10-30%	30-70%
Back-End Bound	20-40%	20-60%	20-40%
Front-End Bound	5-10%	10-25%	5-10%
Bad Speculation	5-10%	5-10%	1-5%

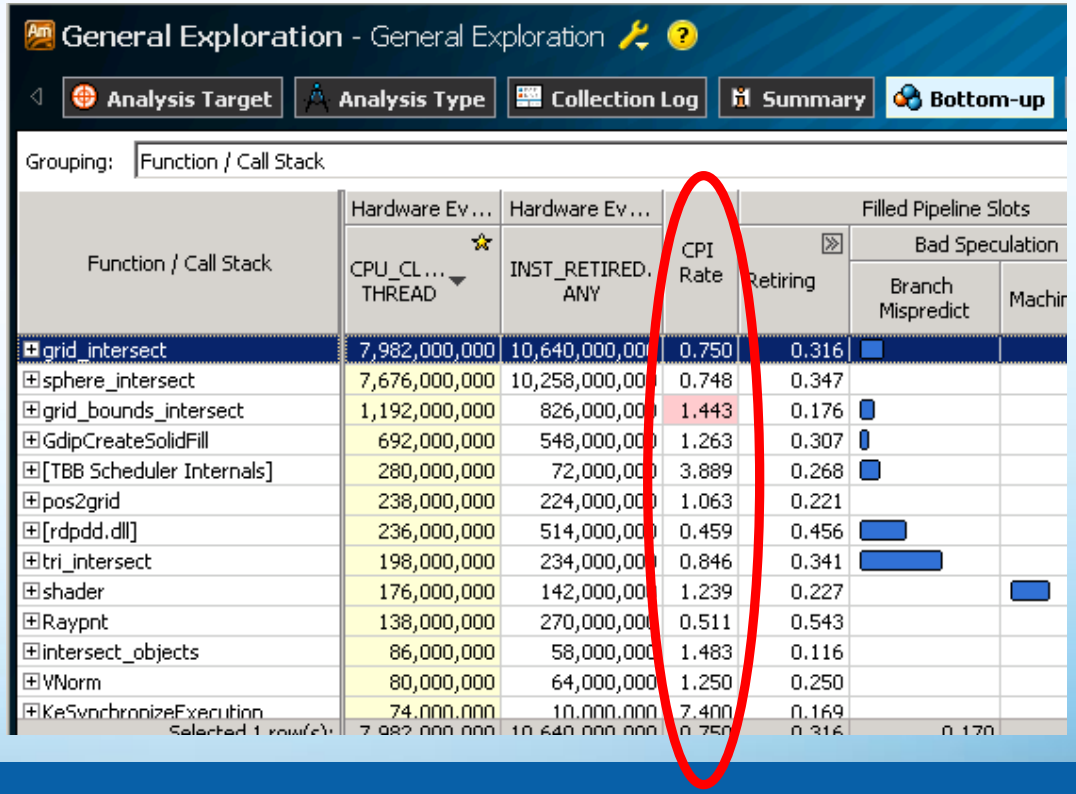
Efficiency Method: % Retiring Pipeline Slots

- **Why:** Helps you understand how efficiently your app is using the processors

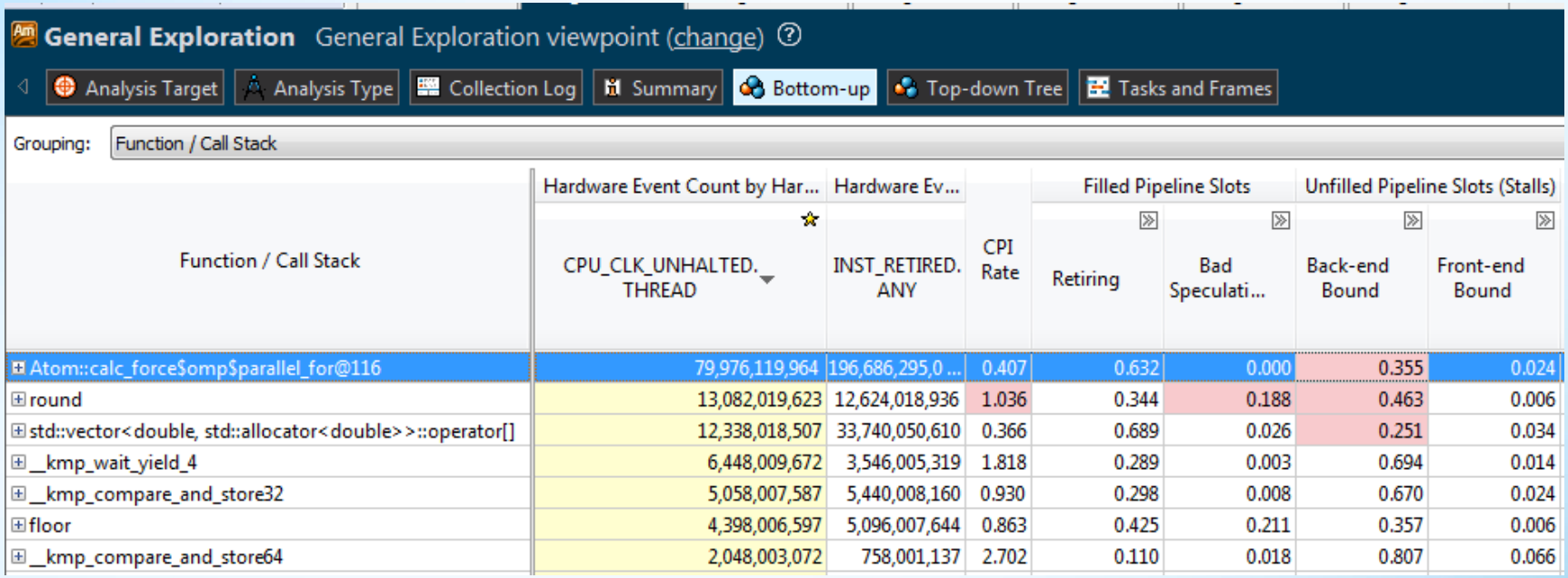


Efficiency Method: Changes in Cycles per Instruction (CPI)

- **Why:** Another measure of efficiency that can be useful when comparing 2 sets of data
  - Shows average time it takes one of your workload's instructions to execute



Microarchitectural Tuning - Top-Down



General Exploration General Exploration viewpoint (change)

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	Hardware Event Count by Har... CPU_CLK_UNHALTED.THREAD	Hardware Ev... INST_RETIRED.ANY	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculati...	Unfilled Pipeline Slots (Stalls): Back-end Bound	Unfilled Pipeline Slots (Stalls): Front-end Bound
Atom::calc_forceompparallel_for@116	79,976,119,964	196,686,295,0 ...	0.407	0.632	0.000	0.355	0.024
round	13,082,019,623	12,624,018,936	1.036	0.344	0.188	0.463	0.006
std::vector<double, std::allocator<double>>::operator[]	12,338,018,507	33,740,050,610	0.366	0.689	0.026	0.251	0.034
__kmp_wait_yield_4	6,448,009,672	3,546,005,319	1.818	0.289	0.003	0.694	0.014
__kmp_compare_and_store32	5,058,007,587	5,440,008,160	0.930	0.298	0.008	0.670	0.024
floor	4,398,006,597	5,096,007,644	0.863	0.425	0.211	0.357	0.006
__kmp_compare_and_store64	2,048,003,072	758,001,137	2.702	0.110	0.018	0.807	0.066

- This code is actually pretty good. High retiring percent.
- Let's investigate Back-End bound

Microarchitectural Tuning - Top-Down

Function / Call Stack	Filled Pipeline Slots		Unfilled Pipeline Slots (Stalls)				
			Back-end Bound				
	Retiring	Bad Speculati...	M. Bo.	Core Bound			
				Port Utilization			
				Cycles of 0 ...	Cycl...	Cycl...	Cycles of 3+ Ports Ut...
Atom::calc_forceompparallel_for@116	0.632	0.000	0.062	0.082	0.000	0.000	0.411
round	0.344	0.188	0.249	0.175	0.000	0.000	0.565
std::vector<double, std::allocator<double>>::operator[]	0.689	0.026	0.049	0.092	0.000	0.000	0.372
__kmp_wait_yield_4	0.289	0.003	0.451	0.536	0.000	0.000	0.852
__kmp_compare_and_store32	0.298	0.008	0.415	0.527	0.000	0.000	0.738
floor	0.425	0.211	0.152	0.126	0.000	0.000	0.464

Core Bound

This metric shows how core non-memory issues limit the performance when you run out of OOO resources or are saturating certain execution units (for example, using FP-chained long-latency arithmetic operations).

Port Utilization

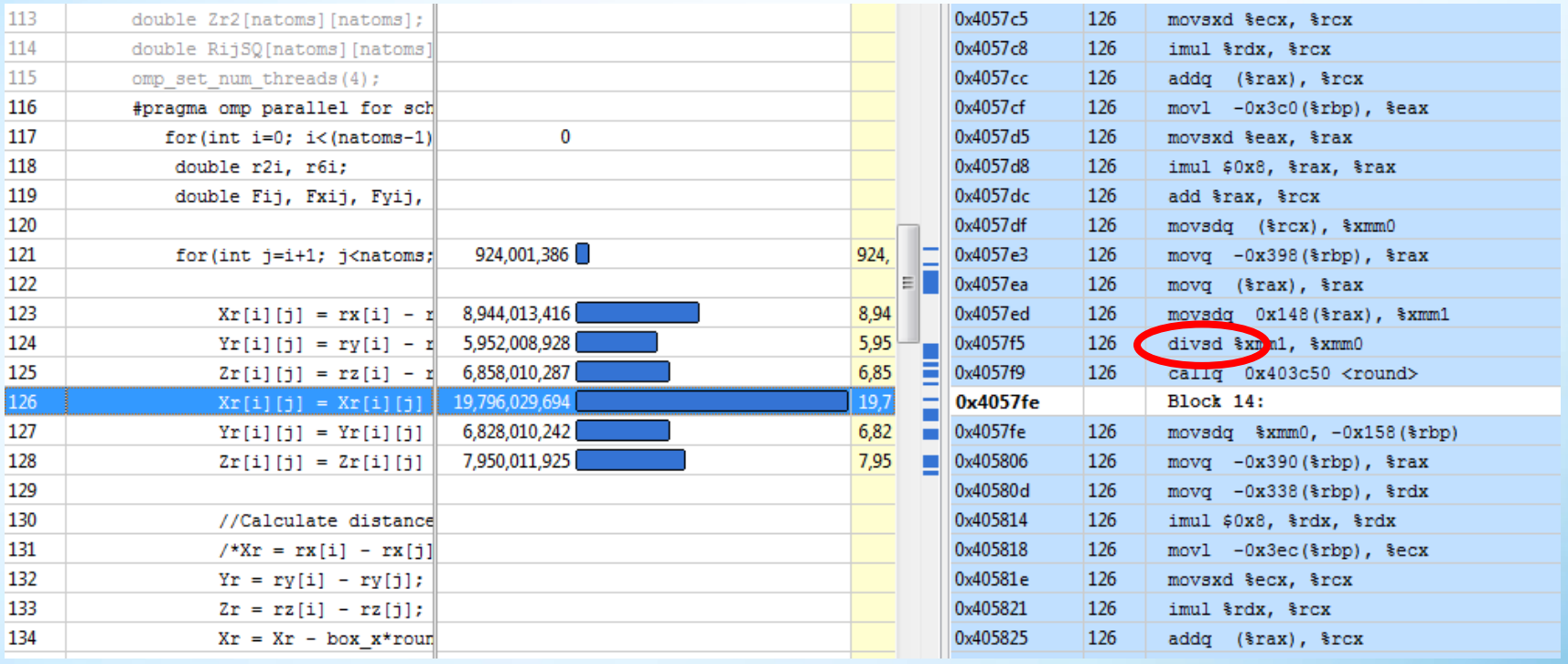
This metric represents a fraction of cycles during which an application was stalled due to Core non-divider-related issues. For example, heavy data-dependency between nearby instructions, or a sequence of instructions that overloads specific ports.

The number of cycles during which 3 or more ports were utilized.

Threshold: ((((UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC) / CPU_CLK_UNHALTED.THREAD) > 0.2) * (CPU_CLK_UNHALTED.THREAD / > 0.05))

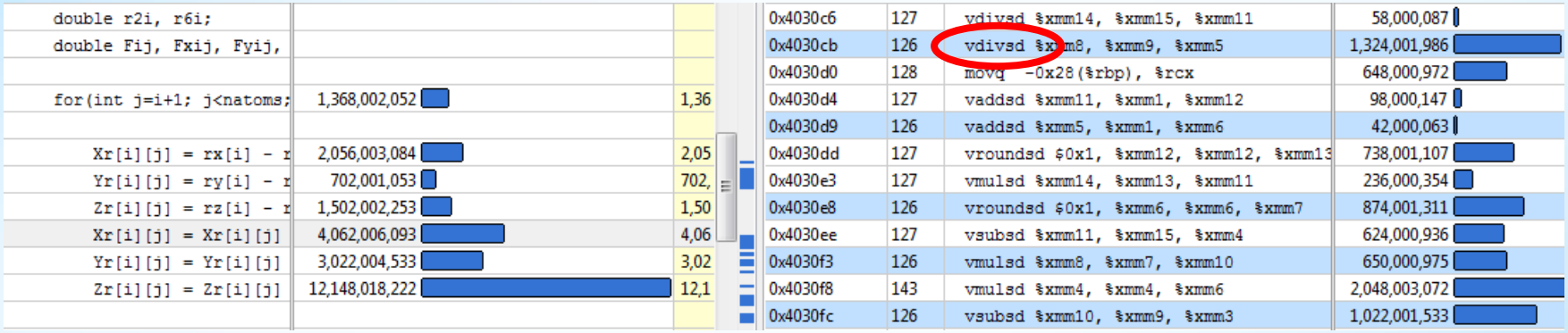
We're basically hammering the compute hardware. Are we vectorizing?

Microarchitectural Tuning - Top-Down



SSE Instructions! Optimize with the compiler e.g. -xhost

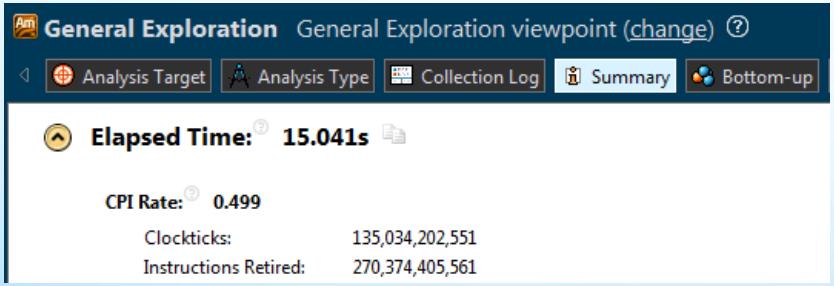
Microarchitectural Tuning - Top-Down



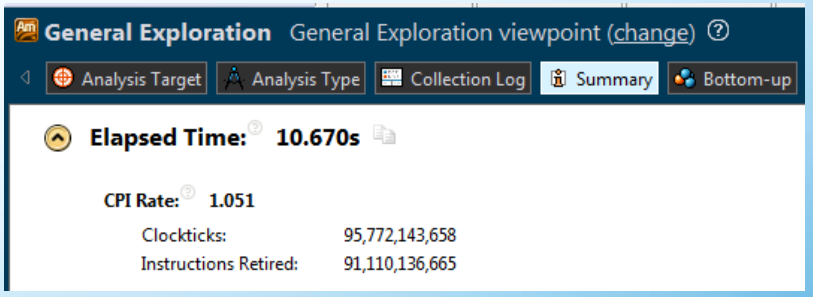
double r2i, r6i;		
double Fij, Fxij, Fyij,		
for(int j=i+1; j<natoms;	1,368,002,052	1,36
Xr[i][j] = rx[i] - r	2,056,003,084	2,05
Yr[i][j] = ry[i] - r	702,001,053	702,
Zr[i][j] = rz[i] - r	1,502,002,253	1,50
Xr[i][j] = Xr[i][j]	4,062,006,093	4,06
Yr[i][j] = Yr[i][j]	3,022,004,533	3,02
Zr[i][j] = Zr[i][j]	12,148,018,222	12,1

0x4030c6	127	vdivsd %xmm14, %xmm15, %xmm11	58,000,087
0x4030cb	126	vdivsd %xmm8, %xmm9, %xmm5	1,324,001,986
0x4030d0	128	movq -0x28(%rbp), %rcx	648,000,972
0x4030d4	127	vaddsd %xmm11, %xmm1, %xmm12	98,000,147
0x4030d9	126	vaddsd %xmm5, %xmm1, %xmm6	42,000,063
0x4030dd	127	vroundsd $0x1, %xmm12, %xmm12, %xmm13	738,001,107
0x4030e3	127	vmulsd %xmm14, %xmm13, %xmm11	236,000,354
0x4030e8	126	vroundsd $0x1, %xmm6, %xmm6, %xmm7	874,001,311
0x4030ee	127	vsubsd %xmm11, %xmm15, %xmm4	624,000,936
0x4030f3	126	vmulsd %xmm8, %xmm7, %xmm10	650,000,975
0x4030f8	143	vmulsd %xmm4, %xmm4, %xmm6	2,048,003,072
0x4030fc	126	vsubsd %xmm10, %xmm9, %xmm3	1,022,001,533

AVX2 on Haswell

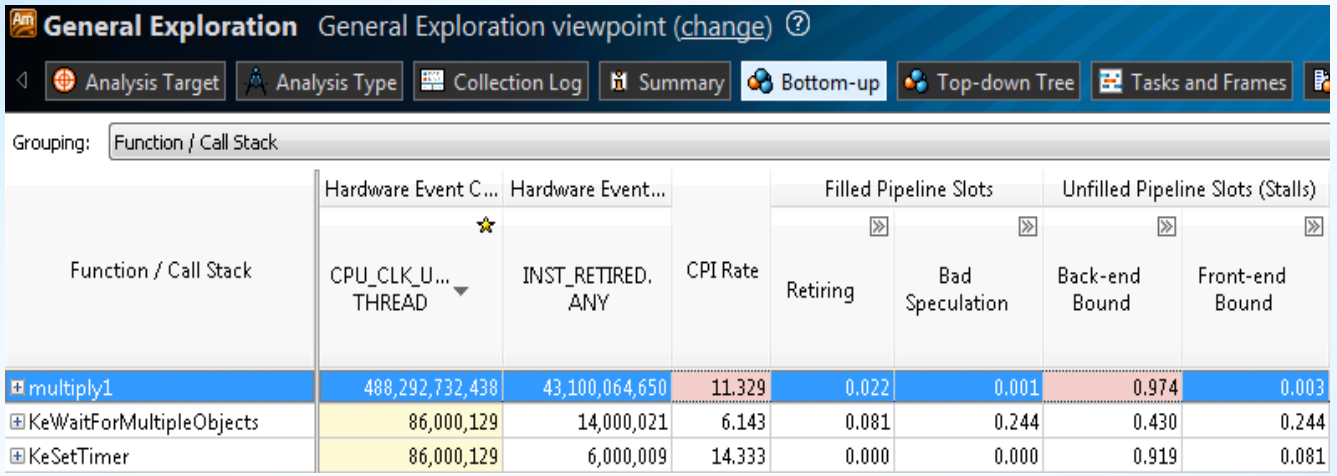


Before

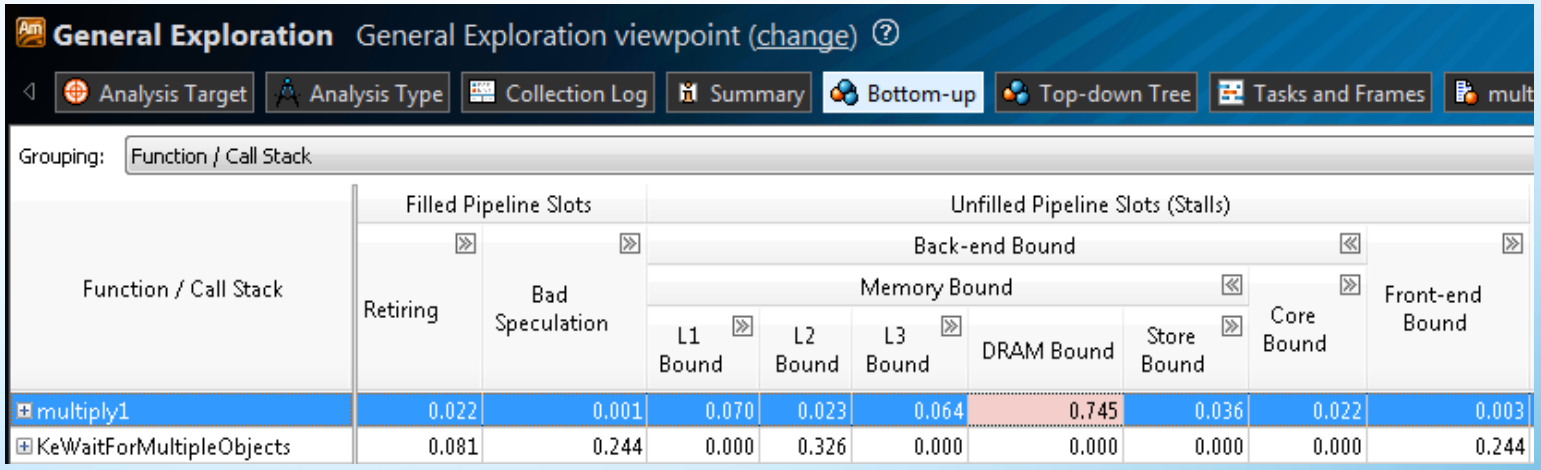


After

Top-Down with a Memory Bound issue



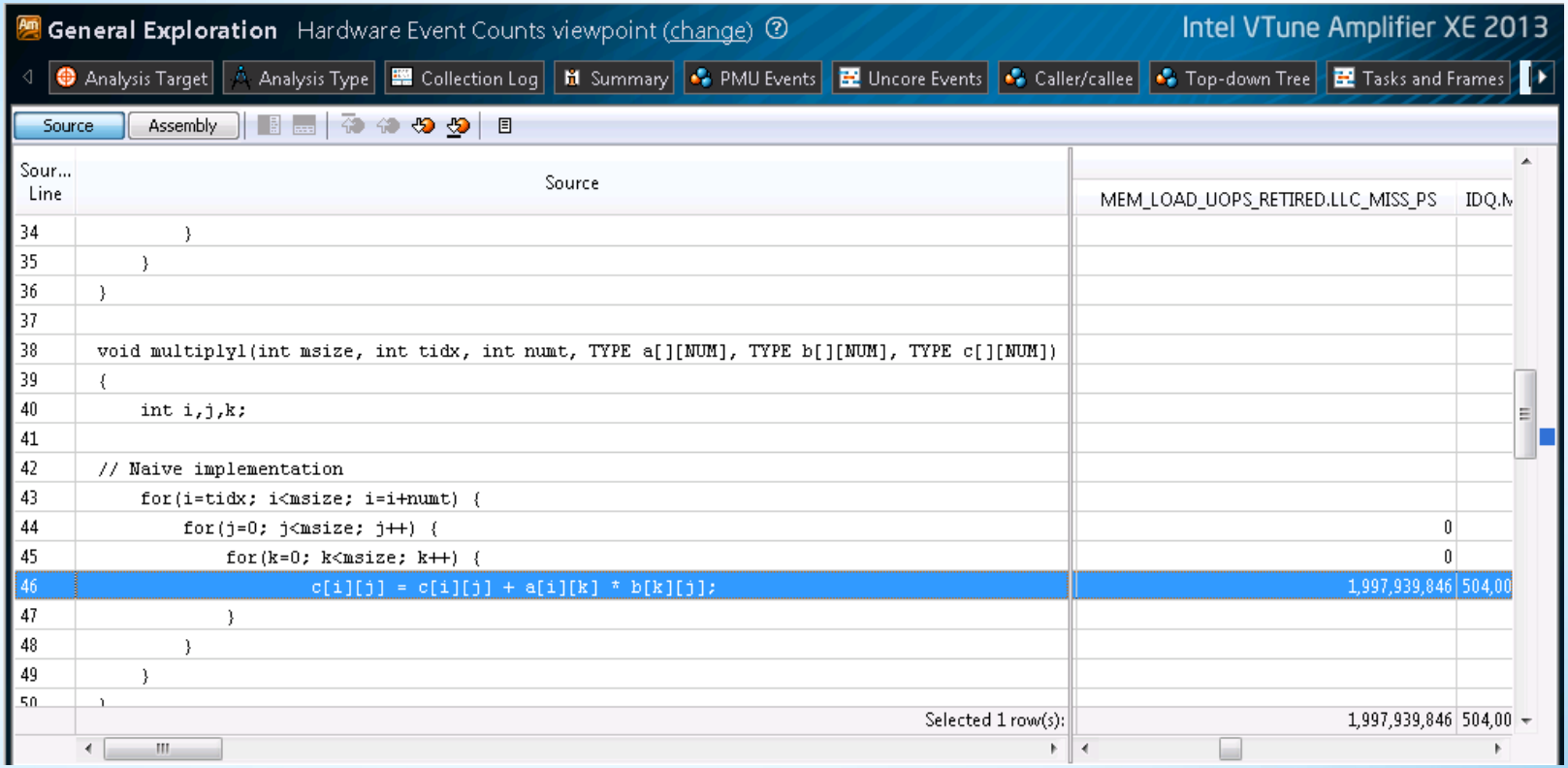
Function / Call Stack	Hardware Event C... CPU_CLK_U... THREAD	Hardware Event... INST_RETIRED. ANY	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculation	Unfilled Pipeline Slots (Stalls): Back-end Bound	Unfilled Pipeline Slots (Stalls): Front-end Bound
multiply1	488,292,732,438	43,100,064,650	11.329	0.022	0.001	0.974	0.003
KeWaitForMultipleObjects	86,000,129	14,000,021	6.143	0.081	0.244	0.430	0.244
KeSetTimer	86,000,129	6,000,009	14.333	0.000	0.000	0.919	0.081



Function / Call Stack	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculation	Unfilled Pipeline Slots (Stalls): Back-end Bound: Memory Bound: L1 Bound	L2 Bound	L3 Bound	DRAM Bound	Store Bound	Core Bound	Front-end Bound
multiply1	0.022	0.001	0.070	0.023	0.064	0.745	0.036	0.022	0.003
KeWaitForMultipleObjects	0.081	0.244	0.000	0.326	0.000	0.000	0.000	0.000	0.244

DRAM Bound Function

Top-Down with a Memory Bound issue

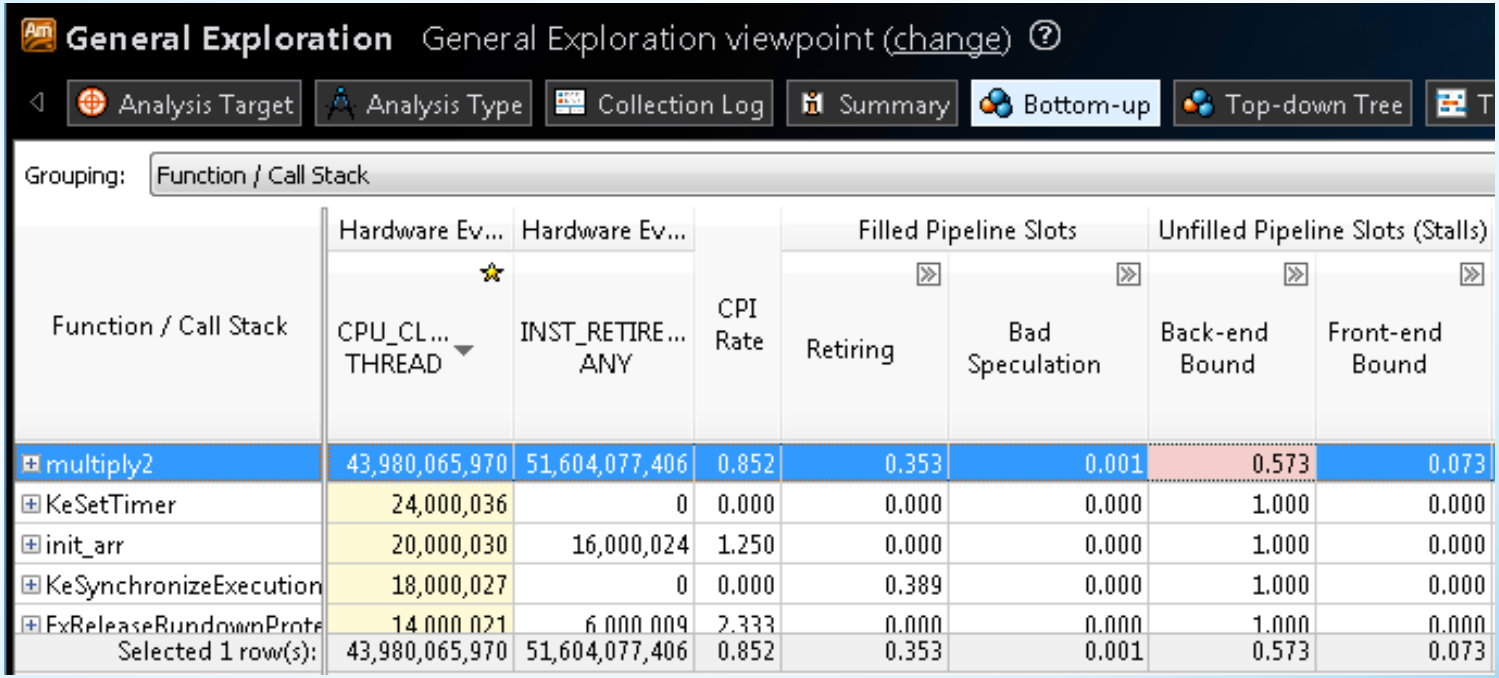


Array accesses are poorly addressed

From Tuning Guide:

- **How:** Memory Bound sub-category, Metrics: *L3 Latency*, *LLC Miss*
- **What Now:**
  - If either metric is highlighted for your hotspot, consider reducing misses:
    - Change your algorithm to reduce data storage
    - Block data accesses to fit into cache
    - Check for sharing issues (See Contested Accesses)
    - Align data for vectorization (and tell your compiler)
    - Use the cacheline replacement analysis outlined in section B.3.4.2 of Intel® 64 and IA-32 Architectures Optimization Reference Manual, section **B.3.4.2**

Top-Down with a Memory Bound issue



Function / Call Stack	Hardware Ev... CPU_CL... THREAD	Hardware Ev... INST_RETIRE... ANY	CPI Rate	Filled Pipeline Slots: Retiring	Filled Pipeline Slots: Bad Speculation	Unfilled Pipeline Slots (Stalls): Back-end Bound	Unfilled Pipeline Slots (Stalls): Front-end Bound
multiply2	43,980,065,970	51,604,077,406	0.852	0.353	0.001	0.573	0.073
KeSetTimer	24,000,036	0	0.000	0.000	0.000	1.000	0.000
init_arr	20,000,030	16,000,024	1.250	0.000	0.000	1.000	0.000
KeSynchronizeExecution	18,000,027	0	0.000	0.389	0.000	1.000	0.000
ExReleaseRundownProte	14,000,021	6,000,009	2.333	0.000	0.000	1.000	0.000
Selected 1 row(s):	43,980,065,970	51,604,077,406	0.852	0.353	0.001	0.573	0.073

With a Loop-Interchange (was 97% Back-End bound)

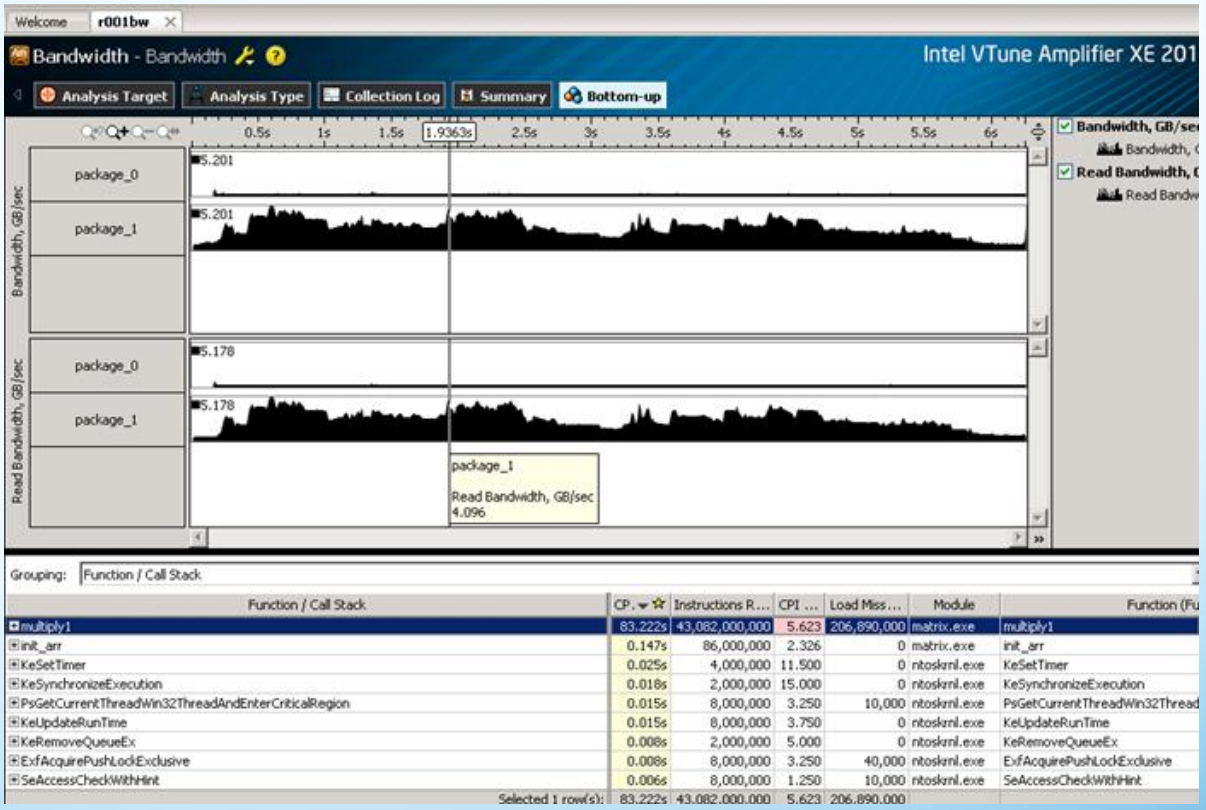
Top-Down for NUMA analysis

Unfilled Pipeline Slots (Stalls)														
Back-end Bound														
Memory Bound													Core Bound	
L1 Bound				Store Bound			L3 Bound			DRAM Bound			DIV Active	Port Utilization
DTLB Ov...	Loads Bl...	Split Loads	4K A...	Fals...	Split...	DTL...	Contest...	Data Shar...	L3 Lat...	Local DRAM	Remote DRA...	Rem...		
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.000	0.000	0.000	0.000	0.267
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
0.099	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.411	0.000	0.000	0.000	0.000	0.283
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.444	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.574

- Multi-socket systems with NUMA require special analysis
  - VTune, numastat, numactl
- Remote cache and DRAM accesses can cause stalls
- Now what?
  - Memory allocation vs. access
  - Temporal locality

Memory Bandwidth using PMUs

- Know your max theoretical memory bandwidth
- Locate areas of high LLC misses
- PMU events available to calculate QPI bandwidth on newer processors



Tuning Guides Have Lots of Metrics and Hints

For example:

Data Sharing

Back-End Bound

- **Why:** Sharing clean data (read sharing) among cores (at L2 level) has a penalty at least the first time due to coherency
- **How:** Memory Bound sub-category, Metrics: *Data Sharing*
- **What Now:**
  - If this metric is highlighted for your hotspot, locate the source code line(s) that is generating HITs by viewing the source. Look for the MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT_PS event which will tag to the next instruction after the one that generated the HIT.
  - Then use knowledge of the code to determine if real or false sharing is taking place. Make appropriate fixes:
    - For real sharing, reduce sharing requirements
    - For false sharing, pad variables to cacheline boundaries

Tuning Guides Have Lots of Metrics and Hints

For example:

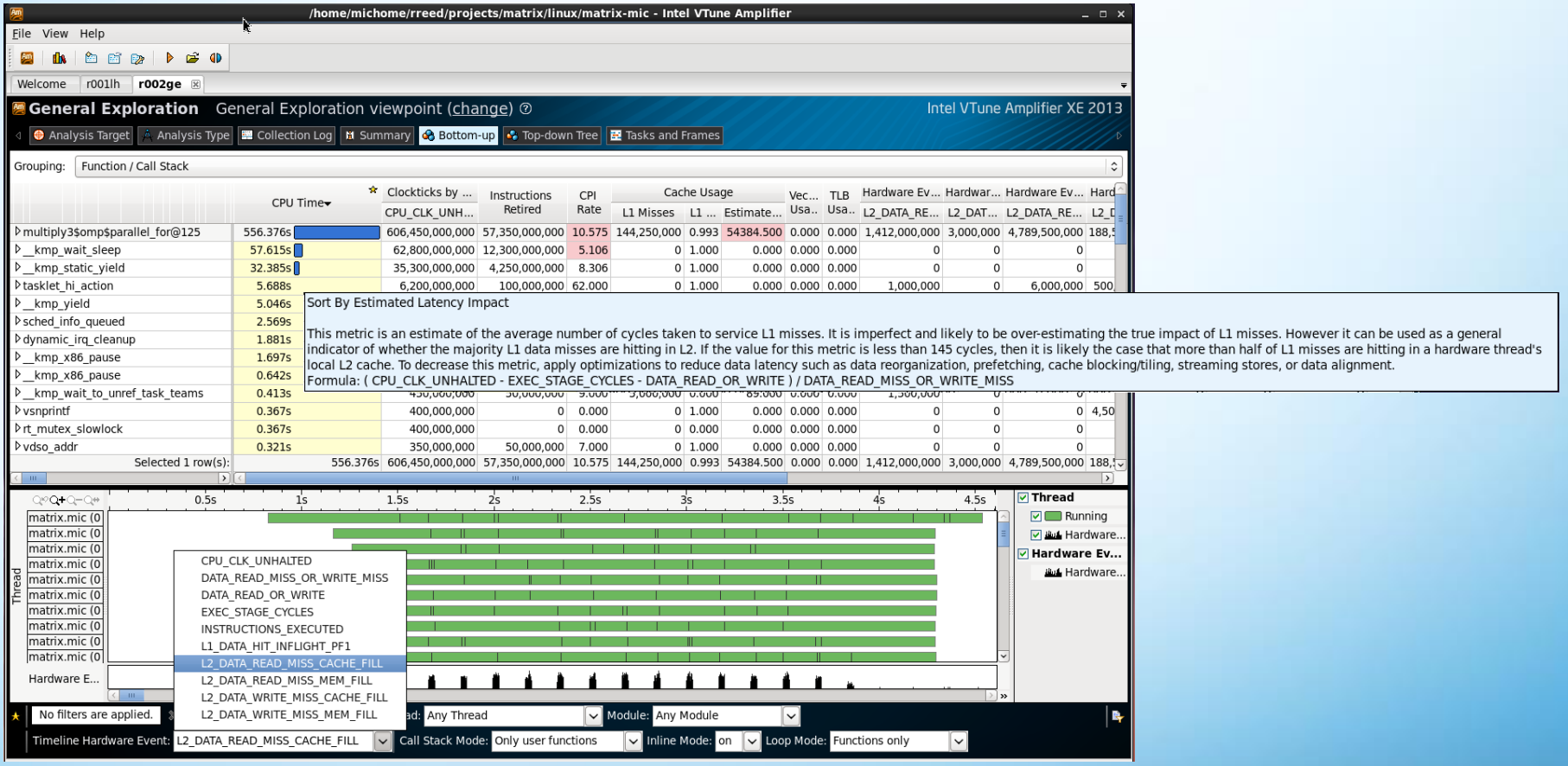
Front-end Latency

Front-End Bound

- **Why:** Front-end latency can lead to the Back-End not having micro-ops to execute (instruction starvation).
- **How:** Front-End Latency sub-category, Metrics: *ITLB Overhead, ICache Misses, Length-Changing Prefixes*
- **What Now:**
  - If any of these metrics are highlighted for your hotspot, try using better code layout and generation techniques:
    - Try using profile-guided optimizations (PGO) with your compiler
    - Use linker ordering techniques (/ORDER on Microsoft's linker or a linker script on gcc)
    - Use switches that reduce code size, such as /O1 or /Os
    - For dynamically generated code, try co-locating hot code, reducing code size, and avoiding indirect calls

Intel Xeon Phi

- Has its own tuning guide and metrics



Intel Xeon Phi

- Efficiency Metric: Compute to Data Access Ratio
  - Measures an application's computational density, and suitability for Intel® Xeon Phi™ coprocessors

Metric	Formula	Investigate if
Vectorization Intensity	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	
L1 Compute to Data Access Ratio	VPU_ELEMENTS_ACTIVE / DATA_READ_OR_WRITE	< Vectorization Intensity
L2 Compute to Data Access Ratio	VPU_ELEMENTS_ACTIVE / DATA_READ_MISS_OR_ WRITE_MISS	< 100x L1 Compute to Data Access Ratio

- Increase computational density through vectorization and reducing data access (see cache issues, also, DATA ALIGNMENT!)

Intel Xeon Phi

- Has its own tuning guide and metrics

- **Problem Area: VPU Usage**
  - Indicates whether an application is vectorized successfully and efficiently

Metric	Formula	Investigate if
Vectorization Intensity	VPU_ELEMENTS_ACTIVE / VPU_INSTRUCTIONS_EXECUTED	<8 (DP), <16(SP)

- Tuning Suggestions:
  - Use the Compiler vectorization report!
  - For data dependencies preventing vectorization, try using Intel® Cilk™ Plus #pragma SIMD (if safe!)
  - Align data and tell the Compiler!
  - Restructure code if possible: Array notations, AOS->SOA

Performance Optimization Methodology

- Follow performance optimization process
  - Use the Top-down approach to performance optimization
  - Use iterative optimization process
  - Utilize appropriate tools (Intel's or non-Intel)
  - Apply scientific approach when analyzing collected results

- Practice!
  - Performance tuning experience helps achieving better results
  - Right tools help as well

Performance Profiling Tools
Technology wise selection

You have a chose of many:

- From simplest and fastest…

Instrumentation Sampling	OS embedded: Task Manager, top, vmstat

- To very complicated and/or slow

Application/platform Simulators	Project embedded: Proprietary perf. infrastructure

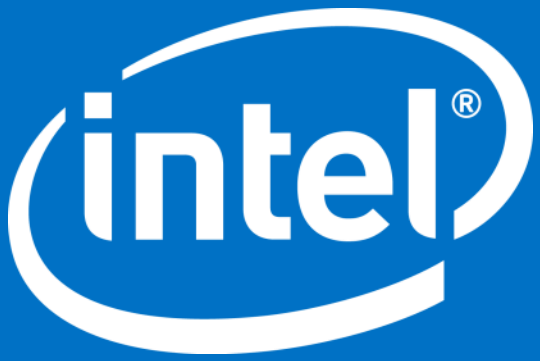
Always consider overhead vs. level of detail – it's often a tradeoff

Scientific Approach to Analysis

- None of the tools provide exact results
  - Data collection overhead or dropping details
  - Define what results need to be precise
- Low overhead tools provide statistical results
  - Statistical theory is applicable
  - Think of proper sampling frequency (for data bandwidth)
  - Think of proper length of data collection (for process)
  - Think of proper number of experiments and results deviation
- Take into account other processes in a system
  - Anti-virus
  - Daemons and services
  - System processes
- Start early – tune often!

References

- Top-Down Performance Tuning Methodology
  - www.software.intel.com/en-us/articles/de-mystifying-software-performance-optimization
- Top-Down Characterization of Microarchitectural Bottlenecks
  - www.software.intel.com/en-us/articles/how-to-tune-applications-using-a-top-down-characterization-of-microarchitectural-issues
- Intel® VTune™ Amplifier XE
  - www.intel.ly/vtune-amplifier-xe
- Tuning Guides
  - www.intel.com/vtune-tuning-guides


intel®

Legal Disclaimer & Optimization Notice

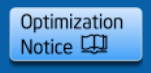
INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804



Copyright © 2012, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.